# LitmusKt: Concurrency Stress Testing for Kotlin

Denis Lochmelis
*Constructor University Bremen*
*JetBrains Research*
Bremen, Germany
dlochmelis@constructor.university

Evgenii Moiseenko
*JetBrains Research*
Belgrade, Serbia
evgeniy.moiseenko@jetbrains.com

Yaroslav Golubev
*JetBrains Research*
Belgrade, Serbia
yaroslav.golubev@jetbrains.com

Anton Podkopaev
*Constructor University Bremen*
Bremen, Germany
*JetBrains Research*
Amsterdam, The Netherlands
apodkopaev@constructor.university

## Abstract

We present *LitmusKt*—the first tool for litmus testing concurrent programs in Kotlin. The tool's novelty also lies in the fact that Kotlin is a *multiplatform* language, *i.e.*, it compiles into multiple platforms, which means that the concurrency has to be tested on several of them. Our tool allows writing litmus tests in a single custom DSL, and these tests are then run in Kotlin/Native and Kotlin/JVM, two main platforms for concurrent programming in Kotlin. Using *LitmusKt*, we discovered novel bugs in the Kotlin compiler, which we then fixed and they are no longer present. Moreover, *LitmusKt* was integrated into the CI pipeline for Kotlin. *LitmusKt* is available on GitHub: https://github.com/JetBrains-Research/litmuskt. The demo is available on YouTube: https://youtu.be/oWCZp_Huwss.

## CCS Concepts

• **Software and its engineering → Software testing and debugging**; **Concurrent programming structures**.

## Keywords

Kotlin, Litmus, Stress Testing, Concurrency, Compiler Design

**ACM Reference Format:**
Denis Lochmelis, Evgenii Moiseenko, Yaroslav Golubev, and Anton Podkopaev. 2025. LitmusKt: Concurrency Stress Testing for Kotlin. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728584

## 1 Introduction

Concurrent programming is notoriously hard and error-prone [14]. In large part, this is due to modern compilers and hardware carrying out various optimizations on programs, such as reordering of instructions and out-of-order execution. Due to these processes, concurrent programs can exhibit counterintuitive behaviors that cannot be observed in the case of single-threaded programs.

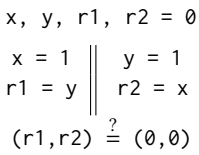



**Figure 1: *Store Buffering* litmus test, a sample program that demonstrates weak behavior.**

As a classic example, consider the *Store Buffering* program in Figure 1, where the program first initializes locations $x$, $y$, $r1$, and $r2$ to 0 and then spawns two threads. Suppose we try to run this program on a single-core machine without compiler optimizations. This would result in some interleaving of read and write operations, as defined by the model of sequential consistency [5]. Notice that both threads start with a write operation either to location $x$ or to location $y$, implying that $r1$ and $r2$ both being equal to 0 at the end of execution is seemingly impossible. However, such behavior can be observed in a real-world multi-core setting due to compilers reordering instructions in threads (*e.g.*, GCC may do so [16]), and such outcomes are referred to as *weak* [1].

The set of allowed outcomes for a concurrent program in a language is usually specified by the language's *memory model* [2, 9]. Developers use the model to reason about their concurrent programs, and compiler creators use it as a specification to which they must adhere. Unfortunately, the latter is not always easy, and there are bugs in implementations of concurrency primitives, *e.g.*, in a popular implementation of JVM [6]. Such bugs in compilers and virtual machines may have a much bigger impact than in user-level applications [6, 12], therefore, they require thorough testing.

In this work, we focus on testing the implementation of concurrency in Kotlin [3], a popular language developed by JetBrains. The fact that Kotlin programs can be compiled to different platforms—JVM, Native via LLVM, JavaScript, and WebAssembly—makes it a particularly interesting target for concurrency testing, since each of these platforms has a different underlying memory model. This in turn requires testing that the behavior of the same Kotlin program

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*FSE Companion '25, Trondheim, Norway*
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728584

on different platforms is uniform. At the moment, Kotlin lacks a formally defined memory model, so there is no solid reference to test the implementation against. However, there are some well-known general patterns that can still be tested and that we decided to discuss with Kotlin's language designers and compiler engineers.

There are different ways to test the compiler's memory model. A popular method is to use the *litmus testing* technique [1, 13]. It works as follows. First, we write specially crafted small concurrent programs, called *litmus tests*, which are designed to highlight specific behaviors or potential bugs. An example of such program, the *Store Buffering* litmus test, is shown in Figure 1. Then, we run them repeatedly for some time and record all the results that were produced by these programs. Finally, we check whether these results are allowed under our assumptions about the language memory model. If we see a forbidden result, there must be a bug with the language implementation—that is, in the compiler.

In reality, the proper litmus testing process is a bit more involved. Crucially, it is not enough to simply rerun the test multiple times, because as mentioned previously, concurrent programs can exhibit weak behaviors that require certain thread interactions and are considerably rare. Therefore, to maximize the chances of observing all possible behaviors, certain techniques should be applied, and to do that, a litmus testing tool should be employed. There exist several well-known tools like that. One is *jcstress* [13], which takes litmus tests written in Java and runs them in order to test the conformance of the JVM to the Java Memory Model (JMM). Another tool is *herdtools/litmus7* [1], which takes litmus tests written in Assembly and checks the memory model of the processor which is running the test. Unfortunately, neither can be reused fully for our case with Kotlin. One key feature of Kotlin is it being *multiplatform*, which means that we need to perform litmus testing with several platforms, and for Kotlin/Native, no existing tool can help us.

To overcome the existing gap in research and practice, we built *LitmusKt*, the first tool designed specifically for litmus testing the Kotlin compiler. Specifically, our tool has the following features.

- Firstly, *LitmusKt* runs litmus tests written in Kotlin and is able to support different platforms (in particular, Kotlin/Native and Kotlin/JVM, as these two offer the most comprehensive support of concurrent programming). If the tool finds a discrepancy between the behaviors on different platforms, this could mean a potential bug in the Kotlin compiler.
- Secondly, *LitmusKt* employs the methods used by other known litmus tools to increase the frequency of rare concurrent behaviors, including the weak ones. Even without comparing different platforms, finding all unusual behaviors on their own proved to be valuable.
- Finally, *LitmusKt* provides a way for declaring new litmus tests in the form of a DSL. The syntax balances simplicity and flexibility for writing arbitrary tests.

Using this tool, we were able to find new bugs in the Kotlin/Native compiler related to unsafe publication of objects, which we discuss in detail in Section 3. Our fix for these bugs was merged into Kotlin [7], and they are no longer present. In addition, our work has been acknowledged by the Kotlin/Native development team, and we integrated *LitmusKt* into the Kotlin CI testing pipeline [8].

```
val StoreBuffering = litmusTest({
    object : LitmusIIOutcome() { var x, y = 0, 0 }
}) {
    thread { x = 1; r1 = y }
    thread { y = 1; r2 = x }
    spec {
        accept(0, 1)
        accept(1, 0)
        accept(1, 1)
        interesting(0, 0)
    }
}
```

**Listing 1: *Store Buffering* test written in *LitmusKt* syntax.**

We believe that our tool is of value to both researchers and practitioners for continuing the study of concurrency in Kotlin, but also broader—for litmus testing in several platforms at the same time and finding novel types of *multiplatform* bugs.

*LitmusKt* is openly available on GitHub: https://github.com/JetBrains-Research/litmuskt. The demonstration video is available on YouTube: https://youtu.be/oWCZp_Huwss.

## 2 Implementation

To highlight how the tool operates, we first demonstrate an example of a *LitmusKt* test and then discuss the implementation details of the framework.

### 2.1 Example of a *LitmusKt* Test

In Listing 1, one can see an example of a litmus test written in the *LitmusKt* DSL syntax (the same *Store Buffering* test presented in Figure 1). The test is declared with the `litmusTest` function and consists of several parts: the definition of a shared state, running code for several threads, and the specification of test outcomes.

The first lambda argument defines the object to be used as a shared state between the threads. This object declares two shared variables: `x` and `y`. This particular state object also inherits from the predefined class `LitmusIIOutcome`[1] that brings into the scope two additional integer variables `r1` and `r2`.[2] Their final state is automatically used as the test outcome. In general, it is possible to either use different `LitmusXXXOutcome`-s, which are optimized for convenience and performance, or declare a completely custom outcome using the `outcome` section of the DSL.

Next, the `thread` DSL sections simply define the running code for each thread. Finally, following the convention used in *jcstress*, the `spec` DSL section specifies what outcomes are

- acceptable—sequentially consistent outcomes, *i.e.*, reproducible as an interleaving of threads;
- interesting—weak but tolerable outcomes according to the expected semantics of the programming language;
- or forbidden—weak intolerable outcomes, indicating bugs.

For example, as mentioned in Section 1, `(0,0)` is a tolerable result in *Store Buffering*, and so it is marked as interesting in Listing 1.

[1] `II` stands for `IntInt`, the naming scheme is taken from *jcstress*.
[2] `rN` stands for *register #N*—by convention in the weak memory literature, registers denote *local variables of threads*, as opposed to *variables shared between several threads*.

| Outcome | Type | Count | Frequency |
|---|---|---|---|
| (1, 0) | ACCEPTABLE | 505,376 | 50.537% |
| (0, 1) | ACCEPTABLE | 493,675 | 49.367% |
| (0, 0) | INTERESTING | 942 | 0.0942% |
| (1, 1) | ACCEPTABLE | 7 | <0.001% |
| total count: 1,000,000, overall status: INTERESTING | | | |

**Table 1: Sample *LitmusKt* output after running the *Store Buffering* test.**

When the test is written, the user can run it using the provided command line interface (CLI). The CLI allows setting various parameters for the run, such as testing duration, number of parallel runs, thread CPU affinity, and so on. All parameters available to the CLI are described in the project's README [4].

Once the test finishes, *LitmusKt* produces a resulting report (see an example in Table 1). The resulting table has a separate row for each encountered outcome: the columns of the table describe the category of an outcome, the number of times it was encountered, and its frequency. The overall status of the entire test is forbidden if at least one forbidden outcome was found, and otherwise it is interesting if at least one interesting outcome was found.

The produced output is meant for manual analysis. For example, as mentioned in Section 1, the *Store Buffering* test can easily produce a weak outcome, and this fact is clearly indicated in the sample output. It can also be seen that the weak outcome is not necessarily the rarest. Let us now describe how *LitmusKt* operates.

## 2.2 Kotlin/Native Test Runner

For Kotlin/Native, no solutions exist, and so we had to develop our own test runner from scratch. To increase the probability of triggering weak behaviors, *LitmusKt* employs a set of standard techniques used by other prominent litmus testing tools, such as *jcstress* [13] and *herdtools/litmus7* [1].

The general principle of *LitmusKt* test runner is illustrated by Figure 2, using the same *Store Buffering* test as an example. First, the runner initializes a large array of test states. Then, it launches several threads, as declared in the litmus test being run. These threads iterate over the array of states and simultaneously execute their code on each state instance. To prevent one thread from significantly running ahead of others, every `N` iterations, threads perform a wait on a barrier to synchronize with each other. Finally, after the threads have finished iterating, the outcomes are collected from the states and quickly analyzed to produce the overall test result.

With this setup, the threads modify different states simultaneously many times over, significantly increasing the likelihood that some concurrent interaction occurs. This general principle is further enhanced by other techniques, *e.g.*, it is possible to directly set CPU affinity for each thread to ensure complex interactions.

## 2.3 Kotlin/JVM Test Runner Based on *jcstress*

The ability of running tests on both Kotlin/Native and Kotlin/JVM and comparing their results was one of the main requirements of *LitmusKt*. At the same time, for Kotlin/JVM compiler, reusing the established tool *jcstress* [13] would be advantageous, as it already implements many advanced techniques and has a proven

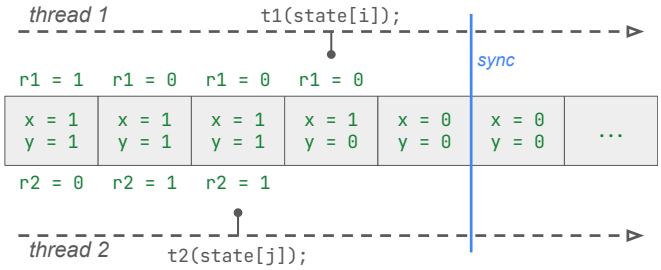



**Figure 2: A schematic of the *LitmusKt* test running process, using *Store Buffering* as an example. One gray rectangle represents a single test state. We allocate an array of them. Then, test threads sequentially execute their functions on this array of states. The threads also regularly synchronize on barriers.**

track record of uncovering concurrency bugs in various JVM implementations [15]. Therefore, we decided to support *jcstress* in the *LitmusKt* as a backend for running tests on Kotlin/JVM.

However, such integration turned out to be technically challenging. The primary issue was that *jcstress* heavily relies on using Java Reflection, a feature not fully supported in Kotlin Multiplatform at the moment. This limitation made it difficult to adapt litmus tests written in our Kotlin syntax for direct use with *jcstress*. Neither was it easy to embed *jcstress* as a library into the *LitmusKt* pipeline, and, specifically, to trigger internal *jcstress* code generation passes.

Because of this, we decided to treat *jcstress* as a "black box" for running tests. The resulting pipeline operates as follows. First, we compile the Kotlin litmus tests into JVM bytecode and pack them into a JAR archive. Second, we use a simple code generator to create a *jcstress*-compliant Java wrapper class for each litmus test. The generated code delegates the execution to the Kotlin-compiled litmus test code from the JAR library created on the first step. Next, *jcstress* runs on these generated Java tests and produces the reports in the HTML format. Finally, we parse these reports into the *LitmusKt* internal format. This entire process is encapsulated within a common test runner interface, enabling seamless integration of the *jcstress*-based test runner in *LitmusKt*.

**Summary.** As demonstrated, *LitmusKt* conforms to all the requirements that we set forth. It allows writing litmus tests in a flexible yet concise syntax, runs them using efficient techniques, and supports multiple platforms by delegating to *jcstress* on JVM and using a custom runner for Native, unifying them under a common interface.

# 3 Analysis

We ran a number of relatively standard litmus tests [11] with *LitmusKt* on two machines: *x86* machine (Intel i7-12700K CPU) and *Arm* machine (Apple M1 CPU). The results are presented in Table 2. Many of the tests that theoretically could show some weak behavior did not do so in practice. This is expected, because Kotlin memory model is not formalized, and as such, it cannot be determined in advance which tests may show weak behaviors in practice. However, some tests still showed noteworthy behaviors, and we will now discuss those tests in more detail.

| Litmus test | Native | | JVM | |
|---|---|---|---|---|
| | x86 | Arm | x86 | Arm |
| ATOM | ACC | ACC | ACC | ACC |
| SB | INT | INT | INT | INT |
| SB volatile | ACC | ACC | ACC | ACC |
| MP | INT | ACC | INT | INT |
| MP volatile | ACC | ACC | ACC | ACC |
| MP DRF | ACC | ACC | ACC | ACC |
| CoRR | ACC | ACC | ACC | ACC |
| CoRR CSE | ACC | ACC | ACC | ACC |
| IRIW | ACC | ACC | INT | INT |
| IRIW volatile | ACC | ACC | ACC | ACC |
| UPUB | ACC | FORB | ACC | ACC |
| OOTA | ACC | ACC | ACC | ACC |
| LB | ACC | ACC | ACC | ACC |
| LB Deps | ACC | ACC | INT | INT |

**Table 2: The results of our analysis. ACC indicates that the results are acceptable, FORB—that at least one result is forbidden, INT—that at least one result is interesting.**

```
class IntHolder(var x: Int = 0)
      h: IntHolder? = null

h = IntHolder()  ||  r1 = h?.x

r1 ∈ {0, null}
```

**Figure 3: The *Unsafe Publication (UPUB)* test.**

Firstly, the *Unsafe Publication* test (*UPUB* for short) demonstrates forbidden outcomes. Its schematic description is shown in Figure 3. Here, one thread initializes a class instance, while another thread simultaneously tries to read its field. When we run this test, we expect to see either no instance in the reading thread (*i.e.*, `null`), or a fully initialized instance, or a partially initialized class with default values for fields. However, as shown in Table 3, when running with Kotlin/Native 1.9.0 on an Arm processor, it is possible to see "garbage" values. The problem is exacerbated if we substitute the `Int` field with a class reference, because when we try to read a reference with a "garbage" address, we get a segmentation fault.

This behavior is clearly not acceptable for a memory-safe language like Kotlin, or in other words, this was a newly discovered bug in the compiler. We investigated it further and then implemented a fix, which was since merged [7]. As of Kotlin 1.9.20 and higher, this bug no longer occurs.

We found more problems regarding *UPUB*, caused by the fact that classes can be published with fields equal to default values. For example, Kotlin has nullable and non-nullable types, and under normal conditions, it should not be possible to read `null` from a non-nullably-typed variable. However, a similar *UPUB with reference* test can be used to observe the opposite, and we do in fact see it. This fact presents a new challenge for language design.

Another similar problem arises from using arrays. In this variation of *UPUB*, one thread initializes an array and another tries to read its first element. In Kotlin, `Array` stores objects, and these objects are also susceptible to the default initialization problem. For example, `Array<Int>` will store boxed `Int` objects, and even when the reader thread is able to read the array element, it can still read a garbage value from them, and again crash with a segmentation fault, which we have confirmed.

| Outcome | Type | Count | Frequency |
|---|---|---|---|
| null | ACCEPTABLE | 26,767,684 | 65.607% |
| 0 | ACCEPTABLE | 14,032,239 | 34.392% |
| 71817408 | FORBIDDEN | 12 | <0.001% |
| -1493360416 | FORBIDDEN | 1 | <0.001% |
| -1560102576 | FORBIDDEN | 1 | <0.001% |
| ... | | | |
| total count: 40,800,000, overall status: FORBIDDEN | | | |

**Table 3: Results of the *UPUB* test, Kotlin/Native 1.9.0, Arm processor.**

To summarize, by running *LitmusKt*, we have discovered several new problems with Kotlin/Native. They have piqued the interest of the Kotlin/Native development team, and it was decided to embed *LitmusKt* as a part of the Kotlin CI testing pipeline [8].

## 4 Future work

During our experiments with *LitmusKt*, we made a couple of observations that suggest some meaningful directions for future work.

One potential path for research is highlighted by the interesting outcomes of the *Message Passing (MP)* litmus test. This test looks similar to *Store Buffering*, even though it tests a different memory model aspect. What is so peculiar about its results in Table 2 is that we observe the weak outcome for Native only on x86 processors, which are generally much more restrictive in terms of weak behaviors compared to Arm. This observation can only be reasonably explained by aggressive compiler optimizations happening due to obscure reasons that cannot be directly controlled. Based on this idea, we believe that fuzzing litmus tests via code mutation to trigger random compiler optimizations has a decent chance of yielding more weak behaviors, and therefore finding more bugs.

Another scope for future work is adding further test runner optimizations, most notably, preventing false sharing. The term *false sharing* refers to a situation when independent variables get allocated on a single CPU cache line, which can easily prevent many weak behaviors. In our internal experiments, we have seen that using some method to avoid false sharing can significantly boost the frequency of weak behaviors, but unfortunately, we struggled to find a general solution to add into *LitmusKt*. We proposed adding a Kotlin/Native annotation similar to JVM's `@Contended` [10].

## 5 Conclusion

We built a litmus testing tool *LitmusKt*, which allows to write litmus tests in Kotlin and run them on different platforms. Using the techniques found in established litmus testing tools, the tool was able to discover new unsafe-publication-related bugs in the Kotlin/Native compiler. The Kotlin/Native team showed interest, and *LitmusKt* has been added into the Kotlin CI testing pipeline. We believe that our tool can be of use to both researchers and practitioners in the further study of the Kotlin's concurrency semantics.

## References


[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 1–74. doi:10.1145/2627752

[2] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL 2011*. ACM, 55–66. doi:10.1145/1925844.1926394

[3] JetBrains. 2024. The Kotlin Programming Language. https://kotlinlang.org/ (Accessed on 2025-04-10).

[4] JetBrains Research. 2024. LitmusKt README. https://github.com/JetBrains-Research/litmuskt/blob/main/README.md (Accessed on 2025-04-10).

[5] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. doi:10.1109/TC.1979.1675439

[6] Shuyang Liu, John Bender, and Jens Palsberg. 2022. Compiling Volatile Correctly in Java. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. doi:10.4230/LIPIcs.ECOOP.2022.6

[7] Denis Lochmelis. 2023. UPUB Fix Merged into Kotlin. https://github.com/JetBrains/kotlin/commit/d2e0361b4e6de7e804c6c079e7195354cd77ced9 (Accessed on 2025-04-10).

[8] Denis Lochmelis. 2024. LitmusKT Merged into Kotlin CI. https://github.com/JetBrains/kotlin/commit/c409279a3833105caf9a4c18c479358c14293ad7 (Accessed on 2025-04-10).

[9] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *POPL 2005*. ACM, 378–391. doi:10.1145/1040305.1040336

[10] Evgenii Moiseenko. 2024. YouTrack Issue about @Contended Annotation. https://youtrack.jetbrains.com/issue/KT-68369/Consider-providing-Contended-annotation (Accessed on 2025-04-10).

[11] Evgenii Moiseenko, Anton Podkopaev, and Dmitry Koznov. 2021. A Survey of Programming Language Memory Models. *Programming and Computer Software* 47 (2021), 439–456. doi:10.1134/S0361768821060050

[12] Jonas Oberhauser, Lilith Oberhauser, Antonio Paolillo, Diogo Behrens, Ming Fu, and Viktor Vafeiadis. 2021. Verifying and Optimizing the HMCS Lock for Arm Servers. In *Networked Systems: 9th International Conference, NETYS 2021, Virtual Event, May 19–21, 2021, Proceedings*. Springer, 240–260. doi:10.1007/978-3-030-91014-3_17

[13] OpenJDK. 2024. Java Concurrency Stress Tests. https://openjdk.org/projects/code-tools/jcstress/ (Accessed on 2025-04-10).

[14] Sergio Rajsbaum and Michel Raynal. 2018. Mastering Concurrent Computing Through Sequential Thinking: A Half-Century Evolution. *arXiv preprint arXiv:1812.06011* (2018). doi:10.48550/arXiv.1812.06011

[15] Aleksey Shipilev. 2024. JCStress Workshop. https://shipilev.net/talks/hydraconf-June2021-jcstress-workshop.pdf (Accessed on 2025-04-10).

[16] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 209–220. doi:10.1145/2676726.2676995